HPC Project: Neural Network Acceleration on GPUs

Team Members: Muaz Ahmed (22i-1125) | Arshaq Kirmani (i220834) | Ibtehaj Haider (22i-0767)

GitHub Repository: https://github.com/arshaqK/Neural-Network-Acceleration-on-GPUs/tree/master (Main for Project Submission)

Presentation: https://docs.google.com/presentation/d/1MFKEoj1a91RdgRyF7rPHOS-PPBuwbGjqRf7nvhiuoKc/edit?usp=sharing

---

Introduction

This report explores the performance acceleration of a simple fully connected neural network on the MNIST dataset using GPU-based parallelization. The goal was to start from a sequential CPU baseline and improve its performance across multiple GPU-accelerated versions, ultimately leveraging CUDA and tensor cores.

Dataset & Model Overview

- **Dataset:** MNIST (28×28 grayscale images, 10 classes, 70k total samples)
- **Model:**
  - Input size: 784
  - Hidden layer: 128 neurons (ReLU)
  - Output layer: 10 neurons (Softmax)
  - Training: Categorical Cross-Entropy + SGD

Version Overview

V1: Sequential CPU Implementation

- **Description:** Pure C-based implementation without any parallelization.
- **Compiler:** GCC with `-O2` and `-pg` flags for profiling.
- **Profiling Tool:** gprof
- **Execution Time:** ~ 97.5s
- **Key Bottlenecks (gprof):**
  - `forward()` : 44.363s
  - `backward()` : 49.039s

Observations:
CPU-bound operations, particularly matrix multiplications and weight updates, dominate time complexity. No use of SIMD or threading. We can see that forward and backward passes take a lot of time and host tasks that can easily be parallelized

---

V2: Naive GPU Implementation (CUDA)

- **Strategy:** Ported forward and backward passes to CUDA kernels with 1D thread blocks.
- **Execution Time:** ~ 22.494s
- **Speedup over V1:** 4.33x

Observations:
The code leverages several advanced CUDA optimization techniques, including tiled matrix multiplications with shared memory to reduce global memory access latency, coalesced memory access patterns, vectorized operations where individual threads process multiple elements simultaneously, and efficient batch processing. Additional optimizations include improved numerical stability in the softmax implementation, proper boundary checking to prevent out-of-bounds memory access, and carefully designed thread block dimensions to maximize occupancy and minimize thread divergence. The implementation also includes comprehensive error checking with the `CUDA_CHECK` macro to catch runtime issues and uses CUDA events for precise performance timing.

The results demonstrate impressive performance gains compared to the previous V2 implementation. While V2 showed training times of approximately 7 seconds per epoch with 96.97% test accuracy, the V3 implementation achieves faster training with times ranging from 0.67 to 0.94 seconds per epoch depending on batch size. When using a batch size of 32 with 8 epochs, the model achieves its best test accuracy of 87.21%, completing the entire training process in just 7.218 seconds. Although the V3 implementation shows slightly lower accuracy than V2, it delivers significantly faster training times, with approximately 7-10x speedup per epoch. The implementation shows that smaller batch sizes (32 vs 64) yield better accuracy despite marginally

longer per-epoch training times, demonstrating that the optimizations effectively balance computational efficiency with model performance.

V3: Optimized GPU Implementation

- **Optimizations Added:**
  - Better launch configurations (block/thread tuning)
  - Shared memory for intermediate values
  - Memory coalescing
  - Reduced memory transfers between host and device
- **Execution Time:** ~ 7.2s
- **Speedup over V1:** 13.54x
- **Speedup over V2:** 3.12x

Observations:
The code leverages CUDA to perform parallel computations across the network's forward and backward passes. Key optimizations include the use of shared memory for matrix multiplications to reduce global memory accesses, tiled matrix operations to exploit locality, vectorized processing where each thread handles multiple elements, improved numerical stability in activation functions, and efficient batch processing. The implementation strategically balances workloads through careful thread block dimensioning and employs synchronization primitives to ensure correct execution order. Error checking is implemented throughout using the `CUDA_CHECK` macro to catch potential runtime issues.

The results shown in the output logs demonstrate the effectiveness of these optimizations. Across various configurations of epochs and batch sizes, the neural network achieves increasingly better performance. With 8 epochs and a batch size of 32, the model reaches its best test accuracy of 87.21%, while maintaining efficient training times. Smaller batch sizes (32 vs 64) appear to yield better accuracy results despite slightly longer training times per epoch. The implementation shows good scalability with the number of epochs, consistently improving accuracy with more training iterations. Total training times remain reasonable, with the 8-epoch training completing in 7.218 seconds, demonstrating that the GPU optimizations effectively accelerate the computation-intensive neural network training process.

V4: Tensor Core Optimization

- **Changes:**

```
- Used Tensor Cores via WMMA API for matrix multiplications

- Converted float32 operations to half-precision where applicable

- Alignment of dimensions to 16x16 tile requirements
```

Limitations:

- Tensor cores require strict layout adherence
- Half-precision impacted accuracy slightly, but remained within tolerable bounds

Observations:
Weren't able to test, however the code written should yield greater results

---

Bonus Version: OpenACC-Accelerated Implementation (Unoptimized)

- **Accelerations Added:**
  - Directive-based parallelization using `#pragma acc` for forward and backward passes
  - Efficient data region management to minimize host-device transfers
  - Loop-level parallelism for matrix operations and activation functions
  - Flattened memory access patterns for better GPU cache utilization
  - Optimized batch-wise processing and fused operations for reduced overhead
- **Execution Time:** ~ 19.98s
- **Speedup over V1:** 4.87x
- **Speedup over V2:** 1.12x

Observations:
This version leverages **OpenACC** to harness GPU acceleration with minimal code rewrites. It provides a clean, portable approach for parallelization, making the neural network train efficiently on GPU while preserving readability. Loop nests in forward and backward propagation were annotated with `#pragma acc parallel loop`, allowing the compiler to generate kernel code automatically. By encapsulating weight matrices and input batches within OpenACC `data` regions, we ensured persistent memory allocation and significantly cut down transfer overhead.

Though it doesn't match the raw performance of hand-tuned CUDA (like V3), the **development speed, maintainability**, and **portability to other accelerators** (like AMD GPUs or multicore CPUs) make this version an excellent trade-off for research, prototyping, or cross-platform applications.

Summary of Results (Example Table)

Version	Execution Time (s)	Speedup vs V1	Accuracy (%)
V1	97.5	1x	97.9
V2	22.494	4.33x	97.9
V3	7.2	13.54x	87.21
V4	N/A	N/A	N/A
V5 (Bonus)	19.98	4.87x	92.53

Conclusion

This project demonstrated how simple neural networks can benefit immensely from GPU acceleration. Starting from a sequential baseline, each version introduced new layers of optimization:

- V2 introduced parallelism,
- V3 tackled memory and kernel launch issues,
- V4 leveraged specialized hardware for maximum speed.

Remaining Bottlenecks:

- Weight updates in backprop still have room for kernel-level parallelization.
- Accuracy vs. precision trade-off in FP16.

Recommendations for Future Work

- Convert to a mini-batch version with shared memory for matrix accumulation
- Add support for multi-GPU training with NCCL or OpenMPI
- Explore cuDNN or PyTorch-C++ for high-level abstraction